# PROJEKAT

## III termin

## Dr Nevena Radović

# Instrukcije skoka

- Za realizaciju struktura koje su opšte poznate u višim programskim jezicima (for i while petlje, if-then-else instrukcije) neophodne su instrukcije skoka u asembleru.
- Asemblerski jezik omogućava upotrebu dva tipa instrukcija skoka:
  - Instrukcije bezuslovnog skoka (unconditional branch/jump),
  - Instrukcije uslovnog skoka (conditional branch)
- **Instrukcija bezuslovnog skoka**:

  **j <label>**
- Ova instrukcija omogućava bezuslovni skok na specificiranu labelu.

# Instrukcije skoka

- **Instrukcije uslovnog skoka** omogućavaju uslovni skok na specificiranu labelu u zavisnosti od ispunjenja postavljenog uslova.
- Koristeći terminologiju viših programskih jezika, ove instrukcije zapravo reprezentuju klasičnu IF instrukciju.
- Standardni set instrukcija uslovnog skoka čine:

  *branch equal, branch not equal, branch less than, branch less than or equal, branch greater than, and branch greater than or equal.*

| Instrukcija | Opis |
|---|---|
| beq <Rsrc>, <Src>, <label> | Skoči na labelu ako su <Rsrc> i <Src> jednaki |
| bne <Rsrc>, <Src>, <label> | Skoči na labelu ako su <Rsrc> i <Src> različiti |
| blt <Rsrc>, <Src>, <label> | Signed instrukcija skoka na labelu ako je <Rsrc> manje od <Src> |
| ble <Rsrc>, <Src>, <label> | Signed instrukcija skoka na labelu ako je <Rsrc> manje ili jednako <Src> |
| bgt <Rsrc>, <Src>, <label> | Signed instrukcija skoka na labelu ako je <Rsrc> veće od <Src> |
| bge <Rsrc>, <Src>, <label> | Signed instrukcija skoka na labelu ako je <Rsrc> veće ili jednako <Src> |
| bltu <Rsrc>, <Src>, <label> | Unsigned instrukcija skoka na labelu ako je <Rsrc> manje od <Src> |
| bleu <Rsrc>, <Src>, <label> | Unsigned instrukcija skoka na labelu ako je <Rsrc> manje ili jednako <Src> |
| bgtu <Rsrc>, <Src>, <label> | Unsigned instrukcija skoka na labelu ako je <Rsrc> veće od <Src> |
| bgeu <Rsrc>, <Src>, <label> | Unsigned instrukcija skoka na labelu ako je <Rsrc> veće ili jednako <Src> |

Instrukcije uslovnog skoka

# Instrukcije skoka

- Navedene instrukcije rade sa 32-bitnim registrima (čak i ako su byte ili halfword vrijednosti smještene u registrima).
- Svaka od naredbi uslovnog skoka može biti modifikovana dodavanjem slova ‘z’ na kraj imena instrukcije. Na taj način se dobija naredba koja vrši poređenje sa nulom (0) bez potrebe korišćenja immediate 0 u instrukciji.
- **Primjer:** Instrukcija

  **bne $t0, 0, loop1**

  se može zapisati kao:

  **bnez $t0, loop1**

- **Primjer:** Napisati program u asembleru izračunavanje sume kvadrata brojeva od 1 do n. Radi provjere uzeti da je n=10.
- **Rješenje:**
- Očekujemo: $1^2 + 2^2 + \cdots + 10^2 = 385$

```
#  Deklaracija podataka
.data
n:                .word 10
suma:             .word 0

#  Text/kod zadatka
.text
.globl main
main:
        lw $t0, n          #
        li $t1, 1          # brojevi od 1 do n
        li $t2, 0          # sum
```

```
Loop:
        mul $t3, $t1, $t1       # broj^2
        add $t2, $t2, $t3
        add $t1, $t1, 1
        ble $t1, $t0, Loop      # ako je broj<=n nastavi s radom
        sw $t2, suma            # smjestanje rezultata
#  Kraj programa
        li $v0, 10          # call code for terminate
        syscall             # system call
.end main
```

# Načini adresiranja

- Adresiranje podrazumijeva specificiranje vrijednosti ili adrese podatka kome se pristupa radi čitanja ili upisa.
- Može uključiti stvarnu vrijednost, ime varijable, ili njenu lokaciju u nizu.
- MIPS arhitektura, koju simulira QtSpim simulator, je 32-bitna, pa su stoga i adrese tipa word, odnosno zauzimaju 32 bita.
- Načini adresiranja:
  - Direktno,
  - Immediate,
  - Indirektno.

# Načini adresiranja

- **Direktno adresiranje**: registar ili memorijska lokacija sadrži konkrente vrijednosti. Npr.:

```
lw $t0, varijabla1
lw $t1, varijabla2
```

- **Immediate adresiranje:** konkretna vrijednost je jedan od operanada. Npr.

```
li $t0, 57          # 57 je immediate adresiran
add $t0, $t0, 57
```

# Načini adresiranja

- **Indirektno adresiranje**: Par zagrada ( ) se koristi da označi ovaj način adresiranja. CPU čita priloženu adresu, a potom ide na tu adresu da pristupi vrijednosti. Indirektno adresiranje zahtijeva više angažovanja CPU od prethodno pomenutih načina adresiranja. Na ovaj način se pristupa elementima niza. Npr.:

```
la $t0, lst       # adresa lst je 32-bitna
lw $s1, ($t0)
```

- Učitavanje naredne vrijednosti, tj. narednog elementa niza:

```
add $t0, $t0, 4
lw $s2, ($t0)
```

- Alternativni način pristupanja sljedećem elementu:

```
la $t0, lst
lw $s2, 4($t0)
```

# Načini adresiranja

- U višim programskim jezicija kompajler vodi računa o tome da je indeks elementa niza legalna vrijednost i da se nalazi u dozvoljenom opsegu. Stoga, ukoliko program pokušava da pristupi indeksu koji se nalazi nakon posljednjeg elemnta niza kompajler će javiti grešku i najčešće pomoći da se mjesto nastanka greške identifikuje (npr. ako pokušamo da pristupimo 7. elementu niza koji ima 5 elemenata).
- Kod asemblerskog jezika ovakav mehanizam zaštite ne postoji. Npr. ako pokušamo da pristupimo 7.om elementu niza koji ima 5 elemenata, vrijednost koja se nalazi na navedenoj memorijskoj lokaciji će biti vraćena bez javljanja greške. Naravno, vraćena vrijednost će biti beskorisna.

- **Primjer:** Napisati program u asembleru za izračunavanje sume i srednje vrijednosti niza 30 integer vrijednosti koji se sastoji od neparnih brojeva od 1 do 60.

- **Rješenje:**

```
#  Deklaracija podataka
.data
niz:            .word 1, 3, 5, 7, 9, 11, 13, 15, 17, 19
                .word 21, 23, 25, 27, 29, 31, 33, 35, 37, 39
                .word 41, 43, 45, 47, 49, 51, 53, 55, 57, 59
duzina:         .word 30

suma:           .word 0
srednja_vr:     .word 0

#  Text/kod zadatka

.text
.globl main

main:
                la $t0, niz         # pocetna adresa niza
                li $t1, 0           # index, i=0
                lw $t2, duzina      # duzina niza
                li $t3, 0           # inicijalizacija suma=0
```

```
Loop:
        lw $t4, ($t0)           # ucitaj niz[i]
        add $t3, $t3, $t4       # suma = suma + niz[i]
        add $t1, $t1, 1         # i = i+1
        add $t0, $t0, 4         # update-uj adresu
        blt $t1, $t2, Loop      # ako je i<duzina, nastavi
        sw $t3, suma            # sacuvaj sumu

# izracunavanje prosjecne vrijednosti
        div $t5, $t3, $t2       # srednja_vr = suma / duzina
        sw $t5, srednja_vr

#Kraj programa
        li $v0, 10
        syscall
.end main
```

- **Primjer:** Napisati program u asembleru za izračunavanje median vrijednosti sortiranog niza 30 integer vrijednosti koji se sastoji od neparnih brojeva od 1 do 60.
- **Rješenje:** Računanje mediana za nizove parne dužine:
  median=(niz[duzina/2]+ niz[duzina/2-1])/2

```
#  Deklaracija podataka
.data
niz:            .word 1, 3, 5, 7, 9, 11, 13, 15, 17, 19
                .word 21, 23, 25, 27, 29, 31, 33, 35, 37, 39
                .word 41, 43, 45, 47, 49, 51, 53, 55, 57, 59
duzina:         .word 30

median:         .word 0

#  Text/kod zadatka

.text
.globl main

main:
                la $t0, niz         # pocetna adresa niza
                lw $t1, duzina      # duzina niza
                div $t2, $t1, 2     # duzina / 2
                mul $t3, $t2, 4     # pretvaranje u Bajt-ove
                add $t4, $t0, $t3 # sabiranje sa pocetnom adresom
```

```
        lw $t5, ($t4)           # niz[duzina/2]
        sub $t4, $t4, 4         # adresa prethodne vrijednosti
        lw $t6, ($t4)           # niz[duzina/2-1]
        add $t7, $t6, $t5       # niz[duzina/2] + niz[duzina/2-1]
        div $t8, $t7, 2         #  / 2
        sw $t8, median          # sacuvaj median
#  Kraj programa
        li $v0, 10
        syscall
.end main
```

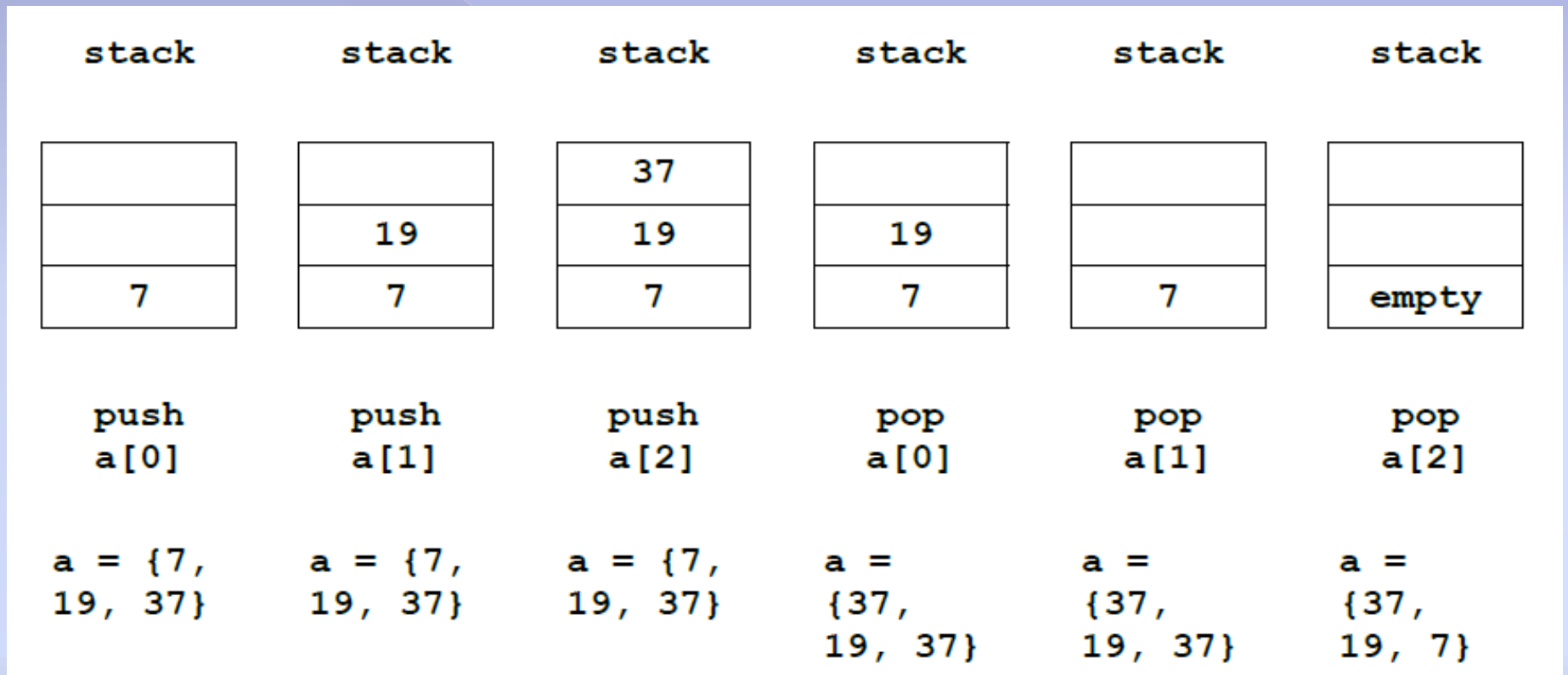
# Stek

- **Stek** je memorijska struktura za skladištenje podataka i intenzivno se koristi prilikom realizacije funkcija, odnosno procedura.
- Podaci se unose na stek, a potom se čitaju u obrnutnom redosljedu. Preciznije, podaci koji se posljednji dodaju na stek se prvi čitaju sa steka. Ovaj princip funkcionisanja se naziva **Last-In, First-Out (LIFO)**.
- Dodavanje podataka na stek se naziva ***push*** operacijom, a uklanjanje podataka sa steka se naziva ***pop*** operacijom.

# Stek

- Posmatrajmo rad steka na primjeru niza a = {7, 19, 37}. Elementi niza a se na stek upisuju operacijama:
  - push a[0]
  - push a[1]
  - push a[2]
- Sa steka se elementi niza preuzimaju operacijama:
  - pop a[0]
  - pop a[1]
  - pop a[2]
- Inicijalna push operacija će učitati 7, potom 19, i na kraju 37. Pošto je stek ***last-in, first-out***, prvi podatak koji će biti preuzet sa steka nakon pop operacije je 37, potom 19 i na kraju 7.
- Uočimo da je na ovaj način došlo do obrtanja redosljeda elemenata niza.

# Stek


stack
stack
stack
stack
stack
stack
7
19
7
37
19
7
19
7
7
empty
push a[0]
push a[1]
push a[2]
pop a[0]
pop a[1]
pop a[2]
a = {7, 19, 37}
a = {7, 19, 37}
a = {7, 19, 37}
a = {37, 19, 37}
a = {37, 19, 37}
a = {37, 19, 7}

# Stek

- Pokazivač na vrh steka je **$sp**, odnosno **$29**.
- Stek se popunjava odozgo prema dolje. Podaci koji se smještaju ili preuzimaju sa steka su **word** formata(32-bita).
- Push i pop operacije nijesu realizovane u asemblerskom jeziku, pa ih je potrebno napraviti pomoću postojećih instrukcija.
- **Primjer:** Smještanje varijable **$t9** na stek:

```
subu $sp, $sp, 4
sw $t9, ($sp)
```

- **Primjer:** Preuzimanje varijable **$t2** sa steka:

```
lw $t2, ($sp)
addu $sp, $sp, 4
```

# Stek

- Vištestruke push/pop operacije:
- **Primjer:** Smjestiti registre $s0, $s1 i $s2 na stek:

```
subu $sp, $sp, 12
sw $s0, ($sp)
sw $s1, 4($sp)
sw $s2, 8($sp)
```

- **Primjer:** Preuzeti varijable sa steka i smjestiti ih u registre $s0, $s1 i $s2 :

```
lw $s0, ($sp)
lw $s1, 4($sp)
lw $s2, 8($sp)
addu $sp, $sp, 12
```